NUMA replicated pagecache for Linux

Nick Piggin
SuSE Labs

January 27, 2008

Talk outline

I will cover the following areas:

- Give some NUMA background information
- Introduce some of Linux's NUMA optimisations
- Show a problem with Linux's NUMA memory allocation
- Introduce the Linux pagecache
- Show how NUMA replication can be applied to the pagecache
- Outline the design of NUMA replicated pagecache

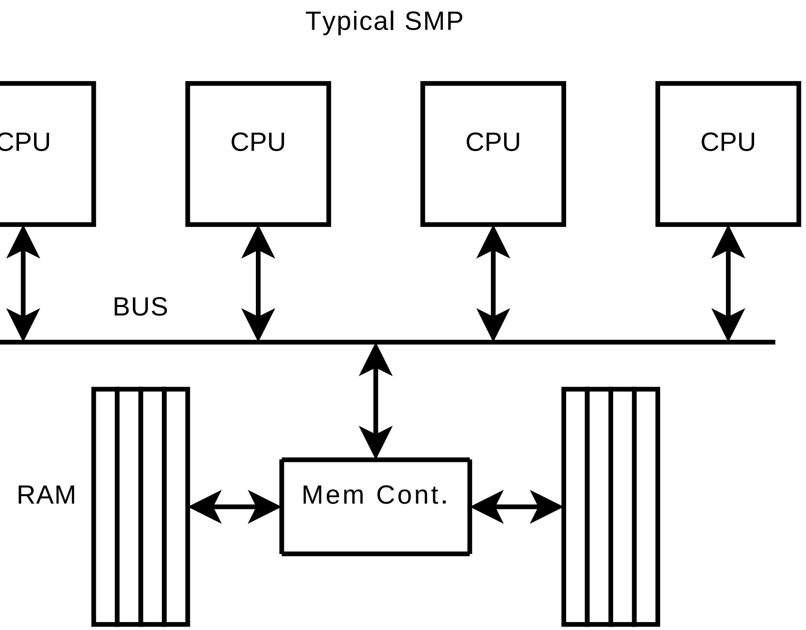
Motivation

Trends show that NUMA is pushing down into commodity systems; also, Linux is pushing up into larger systems.

Scalability remains important for Linux (and software in general). Scalability is the ability to increase throughput of a workload by increasing available CPUs in a multiprocessor system.

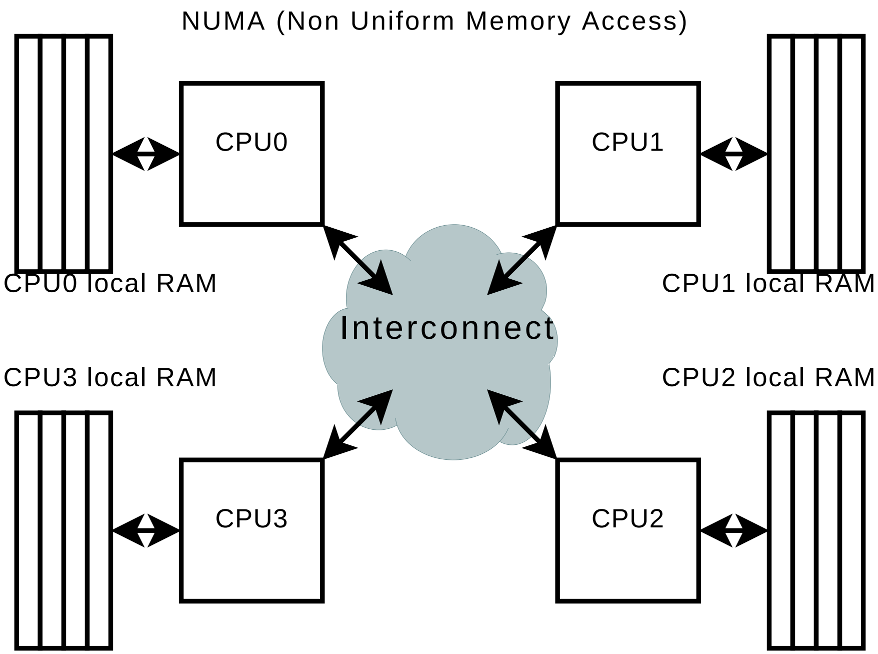
- AMD brought NUMA to the masses, with the Opteron.
- Intel is going to follow soon.
- SGI systems have 512 NUMA nodes, probably more in future.
- All HP, IBM, etc “enterprise” systems are NUMA.

Background



- All CPUs sharing a bus to communicate.
- Typically one memory controller.
- Bus and memory can bottleneck as CPUs are added.

Background cont.

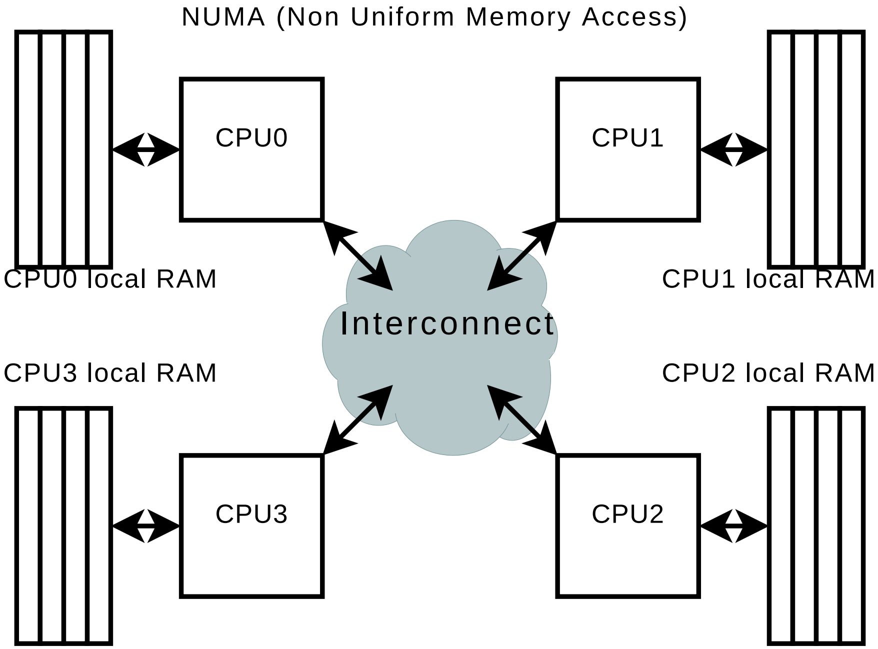


- Each CPU (or group of CPUs) has its own memory controller.
- Each of these groups (CPUs + memory) is called a node.
- All nodes joined together by some interconnect.

NUMA is not a cluster

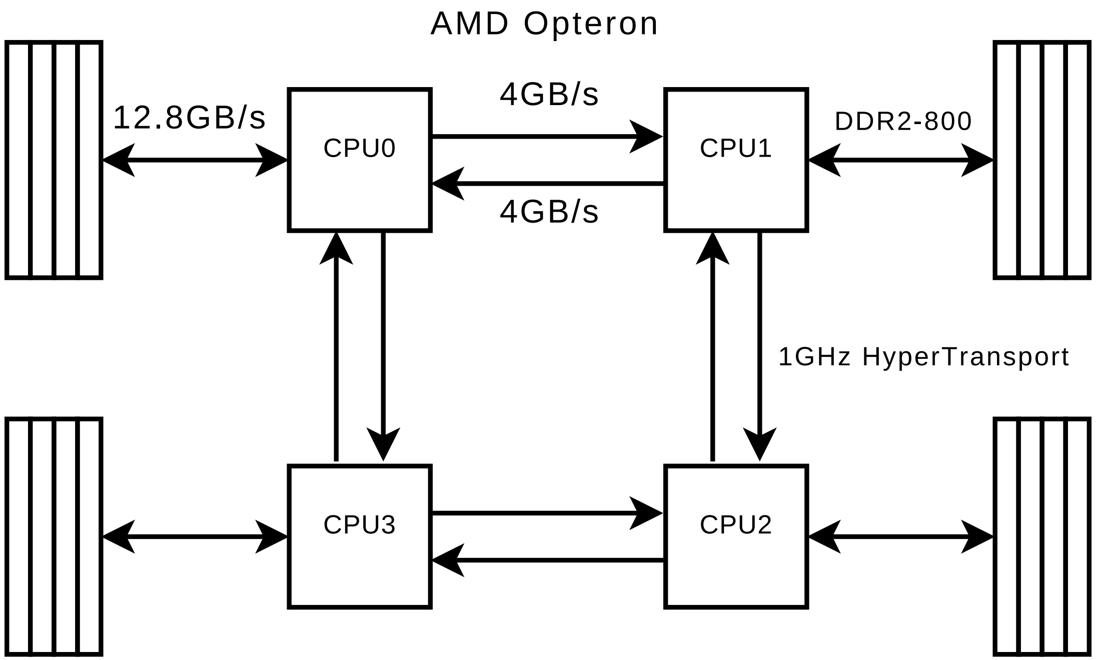
- Definitions vary, but...
- Clusters are not cache coherent (SMP, NUMA are).
- Cache coherency $\approx$ a single system.
- You can run a single operating system image.
- Synchronisation primitives operate on regular memory.
- Can support threads programming models.

Why NUMA?



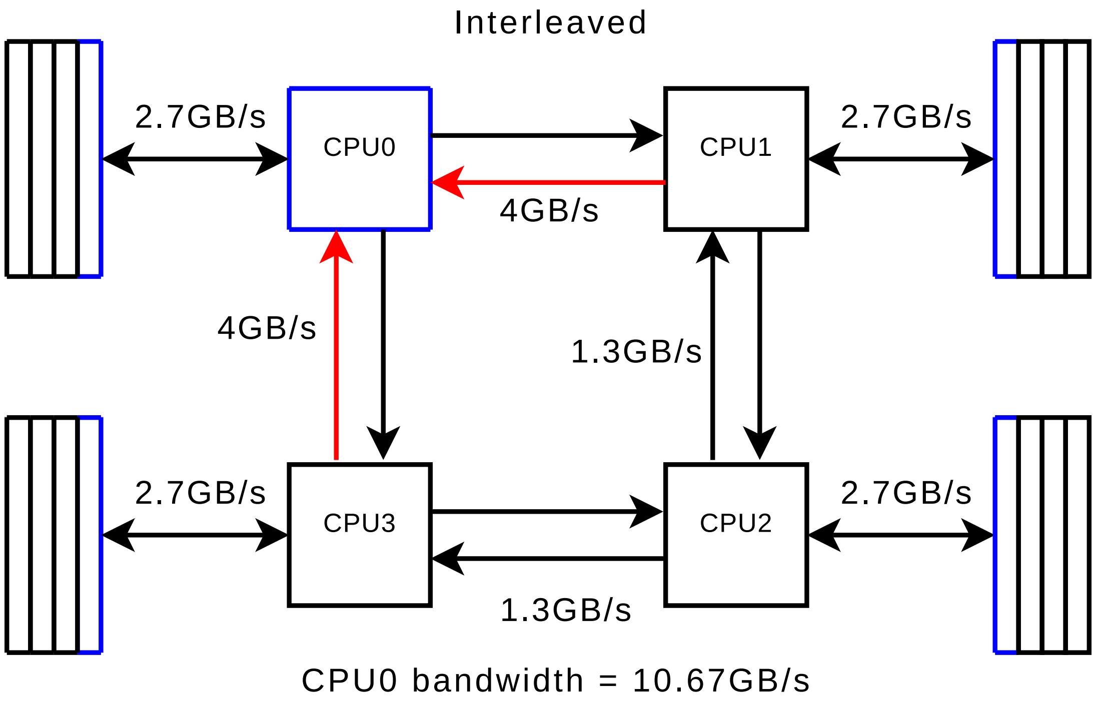
- Access to local memory is very fast (latency and b/w).
- Memory capability scales with CPU count.
- Can support sophisticated cache coherency (BUS cannot).

The AMD Opteron



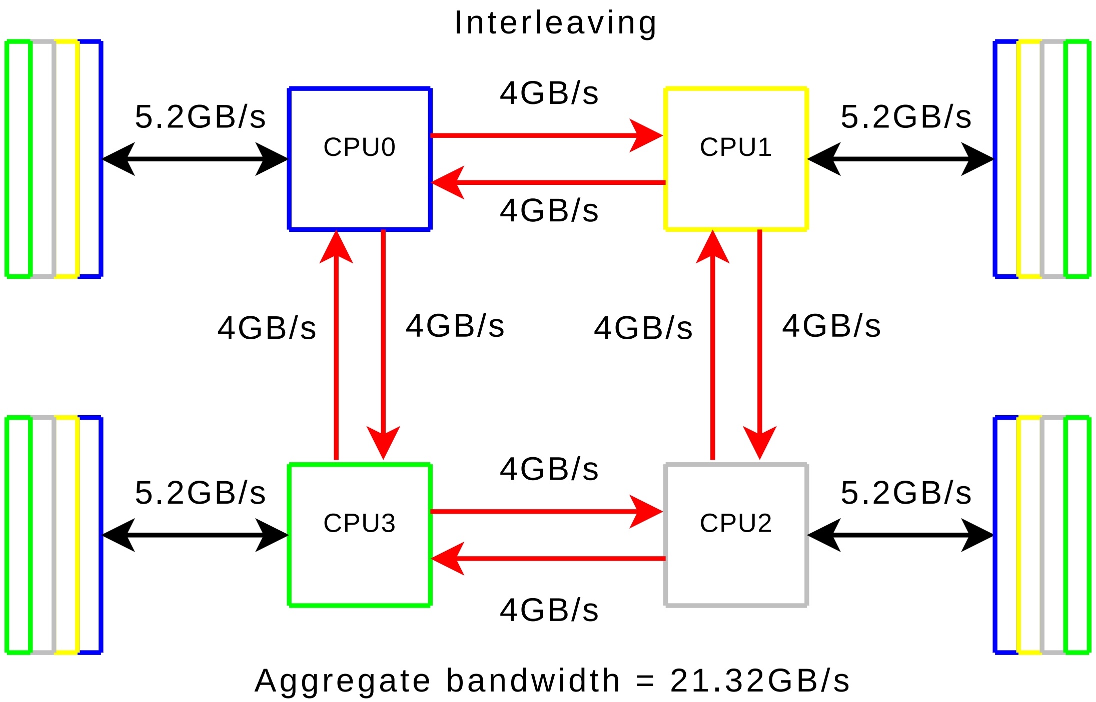
- Common and widely known NUMA architecture.
- ODMC (On Die Mem Controller), becomes NUMA at ≥ 2 sockets.
- Sockets talk to each other via hypertransport interconnect.

Naive NUMA support



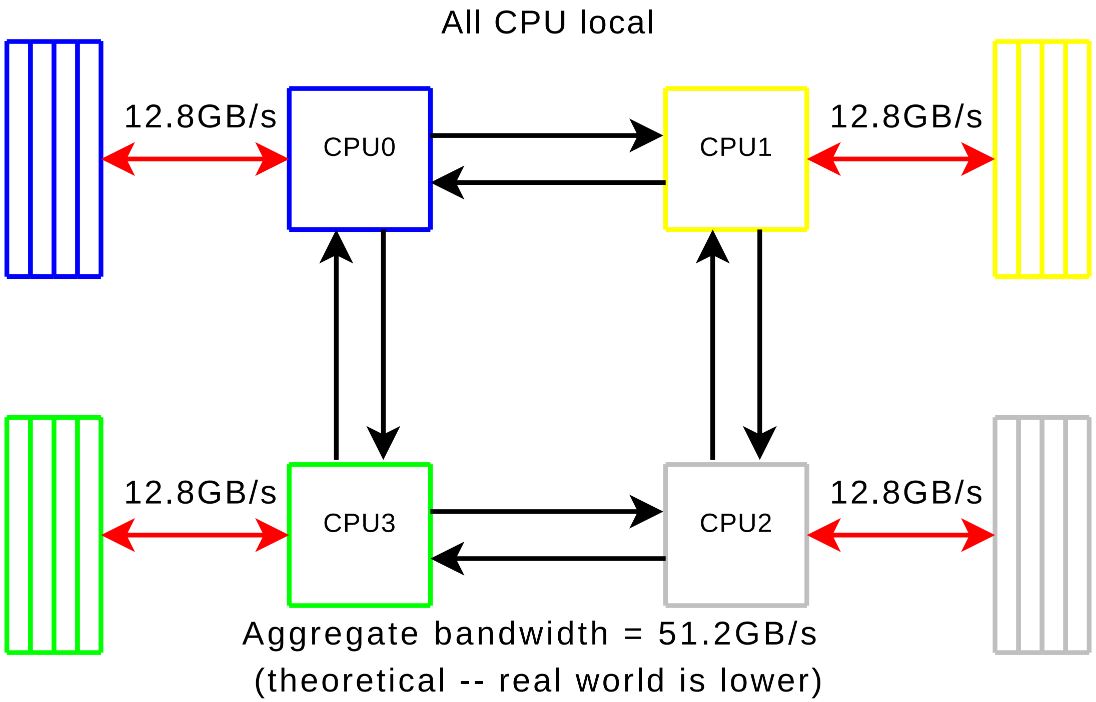
- Interleave allocations across nodes (HW support for non-NUMA OS).
- Statistically evens out the memory load over the system.
- $\frac{nodes-1}{nodes}$ (ie most) memory accesses will be remote.

Naive NUMA support cont.



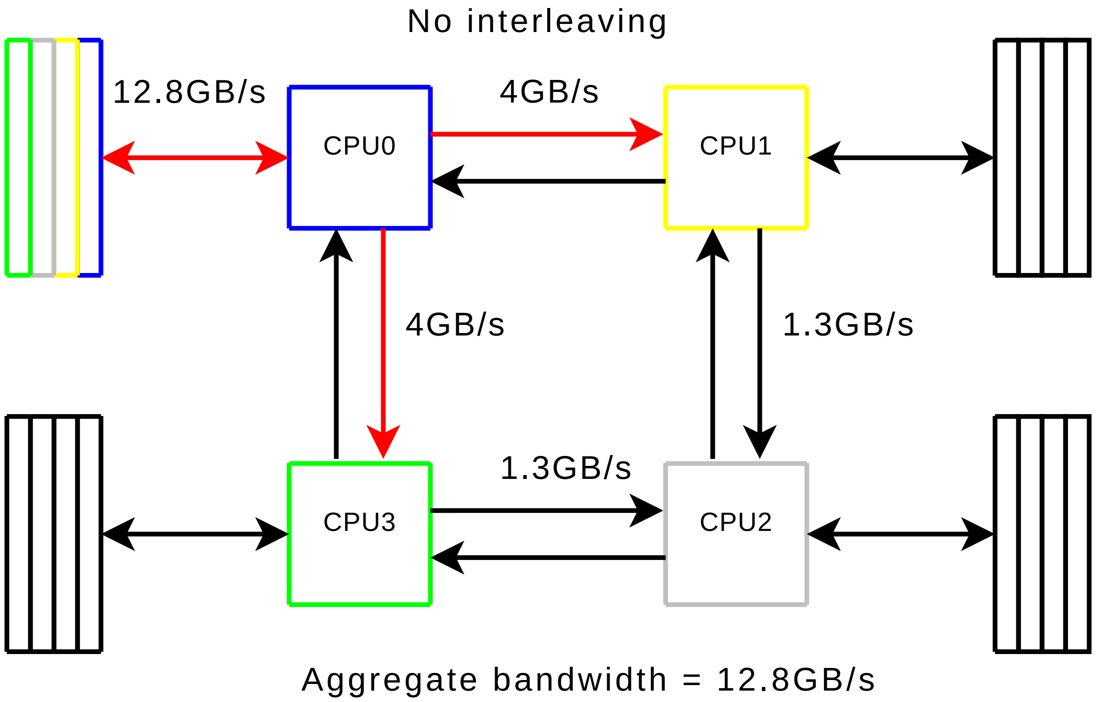
- Interleaving doesn't take advantage of local memory.
- Tends to bottleneck the interconnect.
- Problem only gets worse as system size grows.

Linux NUMA optimisations



- If a process wants memory, allocate from local node by default.
- Works if data is accessed only by that process. Often the case.
- Breaks down if memory is shared between many processes.

When local allocation breaks



- Breaks down if memory is shared between many processes.
- Typical example: shared files in the pagecache (disk cache).
- Worst case: data brought in by one CPU, then used by all.

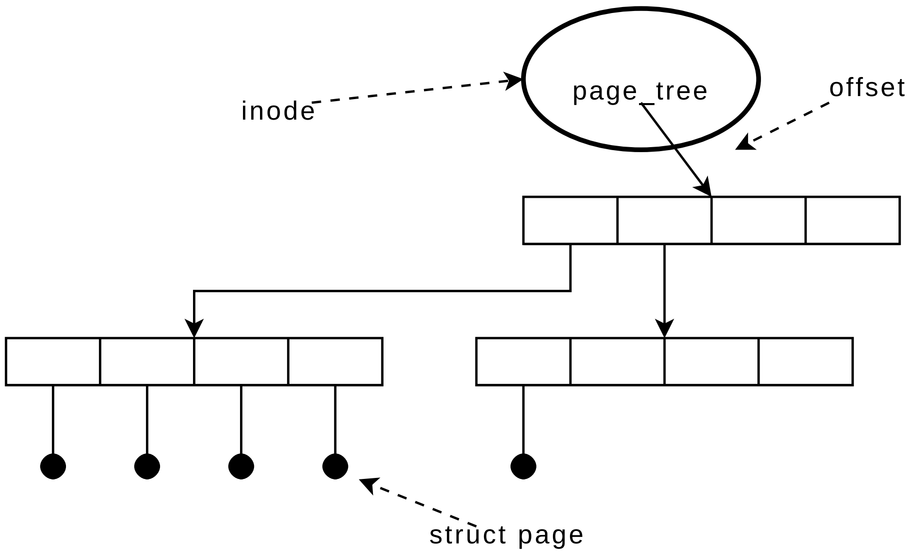
Linux strategy for shared data

- Some pagecache tends to be shared between many processes.
- Shared libs (libc), program text (gcc), shared data (eg. souce code).
- Current strategy: do interleaving for pagecache.
- Not so good if each process works on its own data. It's a tradeoff.

How could we do better?

- For read-intensive data, replication is one possibility.
- Have a copy for each node that is accessing the data.
- If the data is to be read, the process is given the local page.
- If the data is to be written, must first get rid of all but one copy.
- Otherwise, the file looks different when read from different nodes. That's bad.
- Pagecache replication allows read-only shared data to be accessed from node local memory. This is *optimal* memory access!
- It uses more RAM, and it costs a lot to replicate pages and discard them if they get written to. Will require heuristics and "knobs".

Linux 2.6 pagecache intro

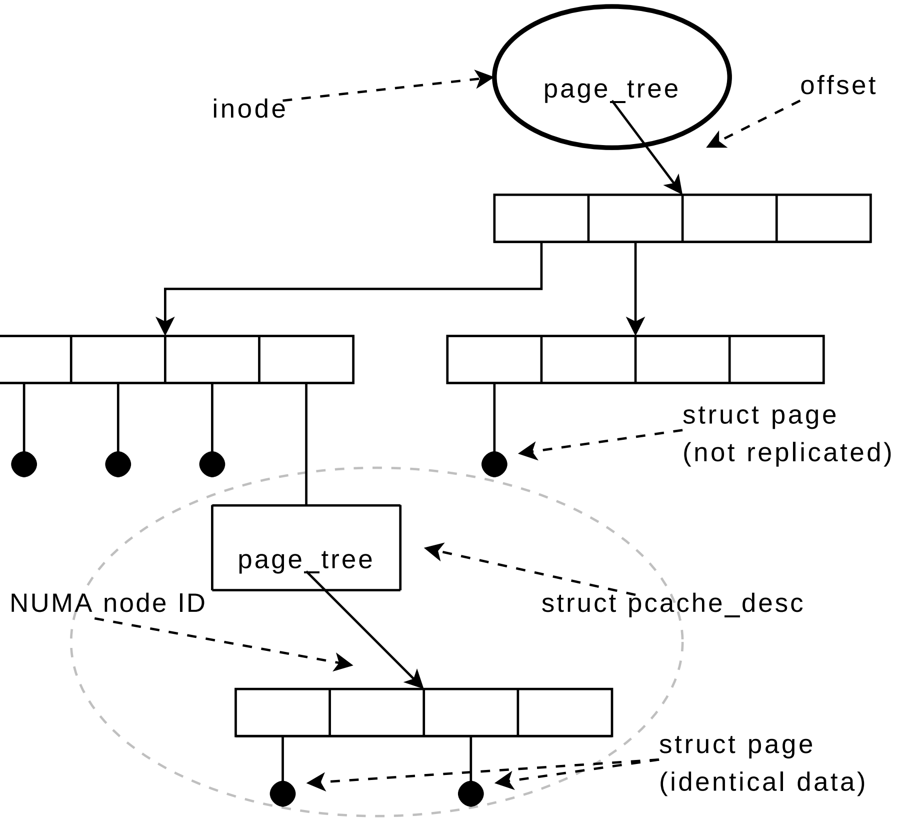


- Linux 2.6 pagecache is a 2-dimensional data structure.
- Used to store or retrieve a page, given `(inode, offset)` key.
- inode contains a radix-tree, keyed by offset, stores page pointers.

Replicated pagecache design

- Perform replication on a per-page granularity.
- Replication occurs opportunistically, at pagecache lookup points in the read(2) syscall, and read-access page fault to an `mmap()`ed file.
- If the page is found but not on the local node, then check if anyone else might write to it. If not, make a local copy (replica), and return that.
- This replica becomes part of the cache, so a subsequent lookup from this node will find the local replica.
- All other paths that look up pagecache first cause all replicas to be removed from any process addresses and discarded (because they might write to the page).

Replicated pagecache design cont.



- Pagecache gains an optional 3rd dimension: NUMA node ID.
- Implemented with another data structure to index pages by node id.

Implementation

Sounds easy, but there are some difficulties!

```
struct pcache_desc {
        struct page *master;
        nodemask_t nodes_present;
        struct page *page_array[MAX_NUMNODES];
};
```

- `struct pcache_desc` – new part of the pagecache data structure.
- `page_array` member can look up a page based on node id.
- `master` is a pointer to a “master” page. This is the one we retain when tearing down other replicas.
- `master` page can contain filesystem metadata.

Pagecache lookups

`find_get_page` is the low level pagecache lookup function.

```
struct page *find_get_page(struct address_space *mapping,
                    unsigned long offset)
{
    struct page *page;

    read_lock_irq(&mapping->tree_lock);
    page = radix_tree_lookup(&mapping->page_tree, offset);
    if (page)
        page_cache_get(page);
    read_unlock_irq(&mapping->tree_lock);

    return page;
}
```

- `radix_tree_lookup` can now return a pointer to *either* a `struct page` or a `struct pcache_desc`, depending on whether or not the page is replicated.
- Must differentiate between the two cases with minimum overhead, I use the lowest bit in the pointer!

Page replication

- Introduce a new function, `find_get_page_readonly`.
- This looks for a local page, and if one can't be found, try to replicate.
- Replication will fail if page is dirty or is referenced from somewhere.
- New function is used by read(2) and page faults (eg. program text).
- `find_get_page`, used everywhere else, must now collapse replicas.
- Write faults to mapped files must also collapse replicas.

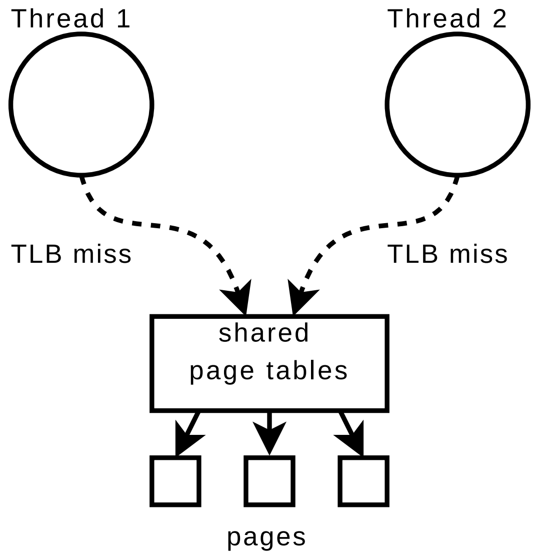
Collapsing replicas

- Remove `struct pcache_desc` 3rd dimension from the pagecache.
- Replace `struct pcache_desc` with “master” page.
- Subsequent references will find the (not replicated) master page.
- Replicas may currently be referenced for read(2), which is OK.
- But pages mapped in page tables must be torn down before writing to the pagecache!

Tearing down pagetables

- Pagecache can be mapped into user address space with mmap(2).
- Replicas allowed in mappings, so long as accesses are read-only.
- But they must all be unmapped when collapsing replicas...
- Next access to memory causes a page fault, maps in the right page.
- Unfortunately, unmapping a page is a complex operation we now must call from `find_get_page` which was previously very simple. Introduces locking dependencies and needs to sleep.

Problem – threads



- Threads all share the same page tables.
- So threads on different nodes cannot all use local replicas.
- Could replicate page **tables** too!
- But let's just worry about processes first.

Problem – process migration

- Process with all its memory on node 1 gets migrated to node 2.
- Continues to run with what is now remote memory.
- Part of a more general problem with Linux NUMA memory allocation.
- Lee Schermerhorn’s automatic page migration aims to solve this.

Code outline

- $\sim$ 700 lines, mostly in mm/replication.c, half a dozen hooks into the core memory manager.
- Configures away without overhead.
- Has a number of global tunables (eg. do not replicate, replicate read-only file descriptors).
- Possibly should have more tunables – eg. cpuset integration, per-process, per-vma policies.
- Needs performance testing. Needs positive results to be considered for upstream.

Performance

- Haven’t been able to get good performance numbers.
- Have not seen any obvious performance regressions.
- Solaris saw about 10% improvement on OLTP on Opteron.

Conclusion

- Pagecache replication could be a good approach to reducing inter-connect traffic and better utilising local memory in NUMA systems.
- There are lots of performance pitfalls (more straight line overhead, more memory copy operations, more RAM used). So replication would have to be used sparingly or in specialised environments.

Thank you